PERFORMANCE OF HYBRID OpenMP/MPI PARALLEL PROGRAMMING APPLICATION OF FINITE ELEMENT METHOD

Leonardo Nunes da Silva, leo.ns@terra.com.br
Flávia Romano Villa Verde, flaviarvv@unb.br
Gerson Henrique Pfitscher, gerson@unb.br
Universidade de Brasília, Campus Darcy Ribeiro, Brasília, DF, Brazil

Abstract. *In the area of parallel processing and parallel algorithms several processors are used together to execute a single application faster. There are two major programming paradigms: Shared Memory and Message Passing. Each of them fits into a specific physical model, but there are multiprocessors architectures whose mapping to one of these paradigms is not so simple. SMP clusters, for example, are built connecting some shared memory machines through an interconnection network. Applications on SMP clusters can be programmed to use message passing between all processors. However, it's possible to achieve better performance using a hybrid model which uses multithreading with shared memory communication inside the SMP node and message passing communication between nodes. The objective of the research presented in this paper was to implement and evaluate a hybrid model of parallel programming for an engineering application in order to evaluate and compare this model with a pure message passing version. Finite element formulation of linear elasticity problems results in linear equation systems that can be solved numerically by the conjugate gradient method. This article presents some numerical simulation results of a structural analysis, where the stiffness matrix of the system is a full matrix, using a 3D parallel code with two strategies for parallelization of the conjugate gradient method. The solution is obtained without any preconditioning technique to provide a performance comparison between a hybrid and a pure version of the same application.*

Keywords*: Parallel Processing, Shared Memory, Message Passing, Hybrid Programming, OpenMP*

1. INTRODUCTION

In the area of parallel processing, several processors are used together to execute a single application faster. There are two major programming paradigms: shared memory and message passing (Culler et al, 1999). The shared memory programming model targets a shared memory architecture, in which multiple processors share single memory space. The communication between processors takes place through reading and writing data in this memory space (Paramount Group, 2005). In the distributed memory programming model, processors don't share memory physically. To communicate, processors need to send messages through the interconnecting network. Applications in the distributed memory model are programmed using libraries of functions. These libraries include functions for sending and receiving messages, synchronizing execution and partitioning data. The Message Passing Interface (MPI) is an important standard that is implemented in the form of such libraries (MPI Forum, 1993).

Applications in the distributed memory model can be programmed using the OpenMP standard. On OpenMP, programmers insert "directives" into the code. These directives do not affect the program semantics. They dictate how work and data shall be shared by the parallel processors. The source code is then compiled by a compiler which provides support for the model and it generates code that creates threads to run on several processors (Dagum and Menon, 1998). Each one of these models fits into a specific physical model, but there are multiprocessors architectures whose mapping to one of these paradigms is not so simple. SMP clusters, for example, are built connecting some shared memory machines through an interconnection network. Applications on Symmetric Multi-Processing, SMP, clusters (Gropp and Lusk, 1995) can be programmed to use message passing between all processors. However, it's possible to achieve better performance using a hybrid model with shared memory communication inside the SMP node and message passing communication between nodes. A hybrid programming model is defined as a model which uses multithreading for shared memory inside the node and message passing between SMP nodes (Chow and Hysom, 2001) (Krawezik and Capello, 2003).

Much work was done in order to verify the gains obtained with the pure message passing models (MPI and PVM lybraries) for a finite element code with superlinear speedups (Villa Verde and Pfitscher, 2005) (Villa Verde et al, 2004a, 2004b, 2004c, 2005, 2006, 2007). In fact, problems modeled by Finite Element Method present high computational costs in terms of times execution and memory use, mainly due to great amount of data to be processed during the solution of the system of equations. For these reasons, we decide for study a hybrid parallelization for this code in order to verify limitations and gains for other parallel programming model. The objective of the research presented in this paper was to develop and evaluate a hybrid model of parallel programming for an application based on the finite elements method. Besides that, provide a performance comparison between a hybrid and a pure version of the same application.

2. PARALLEL PROGAMMING MODELS

Parallel programming models exist at the highest level of a distributed computing system as an abstraction above the parallel hardware and operating systems. Current models can be subdivided into three categories: the shared memory model, the message passing model, and the data parallel model. The shared memory model is based on the concept of single piece of memory that allows shared access between multiple processors or processes. In message passing, the data exchange between the individual processors is performed solely by sending and receiving of messages, where corresponding communication routines are made available via standardized library calls. From a programming point of view, shared address and message-passing programming are perhaps the two most widely used models in contemporary parallel computation. OpenMP is a shared memory model designed as an API for multithreaded applications. Threads communicate by sharing variables (Schäfer, 2006), (Culler et al, 1999).

2.1. MPI

The message passing Interface is a standard for writing message-passing programs. The goal of MPI is to provide standard library of routines for writing portable and efficient message passing programs. MPI provides a rich collection of point to point communication routines and collective operations for data movement, global computation and synchronization (MPI Forum, 1993).

An MPI application can be visualized as a collection of concurrent communication tasks. A program includes code written by the application programmer that is linked with a function library provided by the MPI software implementation. Each task is assigned a unique rank within a certain context: an integer number between 0 and n-1 for an MPI application consisting of n tasks. These ranks are used by the MPI tasks to identify each other in sending and receiving messages, to execute collective operations and to cooperate in general. MPI tasks can run on the same processor or different processors concurrently (El-Rewini and Lewis, 1997).

2.2. OpenMP

OpenMP is a set of compiler directives and callable runtime library routines to express shared memory parallelism (Dagum and Menon, 1998). OpenMP is ideal for developers who need to quickly parallelize existing scientific code without rewrite them. It also can be used to rewrite an entirely new application. The programmer includes directives on the code instructing the compiler how to divide data and computation among processors. Then, when the code is compiles, the compiler generates code to be executed in parallel by the threads.

An OpenMP program executes according to a fork-join model (Fig. 1). This means that an OpenMP program begins execution as a single sequential thread, the master thread. When a parallel directive is encountered by that thread, execution forks and the parallel region is executed by a team of threads in parallel. When the parallel region is finished, the threads in the team synchronize at an implicit barrier, and the master thread is the only thread that continues execution. Figure 1 shows the OpenMP execution model (Hoeflinger, 2006).

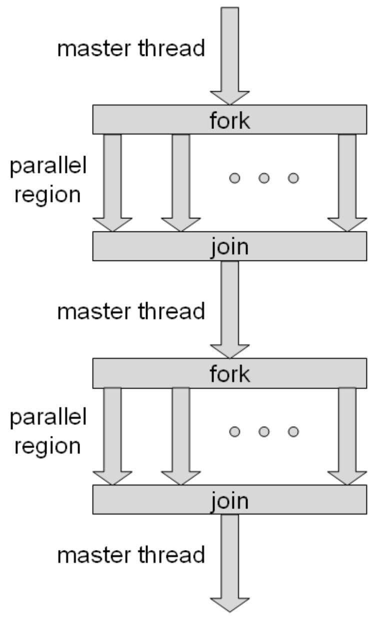


Figure 1. OpenMP execution model

2.3. HYBRID OpenMP/MPI PROGRAMMING

In (Smith and Bull, 2000), a hybrid programming model is defined as a model which uses multithreading for shared memory inside the node and message passing between SMP nodes. By using a hybrid programming model we should be able to take advantage of the benefits of both shared memory and message passing models. Hybrid programs should give better performance than pure message passing ones for three reasons: 1) message passing within a node is replaced by fast shared memory access; 2) there is smaller communication volume on the interconnect since internode's messages are not necessary; 3) fewer processes are involved in communication, which should lead to better scalability, particularly for global communications (Chow and Hysom, 2001) (Capello and Etiemble, 2000).

Applications may or may not benefit from hybrid programming depending on some applications parameters (Rabenseifner, 2003). The advantage is clearly application dependent (Capello and Etiemble, 2000). In (Capello and Etiemble, 2000) has been presented a classification based in the programming efforts: fine-grain and coarse-grain parallelization. The fine-grain approach is the simplest one. It consists in OpenMP parallelization of the loop nests in the computation parts of the MPI code. In this case, is important to choose loops that contribute significantly to the global execution time. In the coarse-grain approach, OpenMP is still used to take advantage of the shared memory inside the SMP nodes but a Single Program Multiple Data, SPMD, programming style. At the beginning of the main program, *N* OpenMP threads are created and each of them acts as it was an MPI process (Capello and Etiemble, 2000).

3. CASE STUDY

Linear elasticity problems modeled by a system of equations generated by the finite elements method can be solved by the conjugate gradient method. With the finite elements method, instead of assuming some properties for the entire body, we divide it into smaller elements and assume these properties for these individual elements. These individual elements are then analyzed and instead of carrying out integration over the entire body we carry out summation over the body consisting of finite number of elements of finite dimensions (Cook, 1995) (Bathe, 1996). The FEM leads to a system of equations of the form:

$$\mathbf{Ax} = \mathbf{b}. \tag{1}$$

Where **A** is a square, symmetric, positive-definite matrix, **x** is an unknown vector and **b** is a known vector. The conjugate gradient method is then applied to iteratively solve the system of equations. The conjugate gradient method consists of a loop over a fixed number of matrix vector and vector-vector operations.

3.1 Message Passing Program Structure

The application input is a mesh for the selected geometry. First, this mesh is partitioned (domain decomposition) using the library METIS (George and Vipin, 1998). The number of partitions is equal to the number of MPI processes. Thus every process handles about the same volume of data. Then every process mount its matrix and some auxiliary data structures based on the partition it receives. And finally a parallel conjugate gradient method is applied to solve the system of equations. During this last step, there is intensive communication between the processes because adjacent elements share nodes on their boundary and they need to exchange information about them during the computation. At the end of the parallel conjugate gradient method, each process has got part of the solution and a reduce operation is performed to create the resulting **x** vector.

The parallel algorithm for the conjugated gradient method is shown in Fig. 2. In this version, the index *i* varies from 0 to the number of tasks minus one and the counter *k* points which iteration is been worked upon. The stiffness matrix **A** must be restructured, with the intention of the values of the products needed to the Conjugated Gradient Method, CGM, solution doesn't have any change. Thus, the matrix **A** of each computing sub-domain is broken in four others $\mathbf{A_p}$, $\mathbf{A_s}$, $\mathbf{B_p}$ e $\mathbf{B_p^T}$ (block-arrowhead form). $\mathbf{A_p}$ is a square matrix with a size equal to the number of interior degrees of freedom (dof) for each sub-domain, where the stiffness values referring to the interior dof are stored. In $\mathbf{A_s}$ are stored the values of the stiffness matrix relating to the dof on the partition boundary.

The $\mathbf{A_s}$ is also square and its size is defined by the number of dof that are located on the partition boundary of each partition. The size of $\mathbf{B_p}$ is defined by the number of degrees of freedom located both on the partition boundary and on the partition's interior. This matrix is used to store the stiffness values that relate the interior dof with the partition boundary ones. $\mathbf{B_p^T}$ is the transpose of $\mathbf{B_p}$. Due to the restructuring of the matrix **A**, the vectors **x** (displacements) and **b** (forces) are also reformulated. Each of them is broken in two parts. The vector **b** generates $\mathbf{b_p}$ and $\mathbf{b_s}$, where the first vector has the size equal to the number of interior degrees of freedom and the other equal to the partition boundary one.

```
For an arbitrary tolerance value ε do
  For $k = 0$
      1. $\mathbf{x}_i^0 = \mathbf{0}$;   $\mathbf{x}_{Si}^0 = \mathbf{0}$;   $\beta^0 = 0$
      2. $\mathbf{b}_{Si} = \text{Update}(\mathbf{b}_{Si})$
      3. $\mathbf{r}_i^0 = \mathbf{b}_i$;   $\mathbf{r}_{Si} = \mathbf{b}_{Si}$
      4. $\mathbf{d}_i^0 = \mathbf{b}_i$;   $\mathbf{d}_{Si} = \mathbf{b}_{Si}$
      5. $\gamma^0 = \text{InnerProduct}(\mathbf{r}_i^0, \mathbf{r}_{Si}^0 \;;\; \mathbf{r}_i^0, \mathbf{r}_{Si}^0)$
  Repeat  steps 6 to 17 until                                   // loop
      6. $\mathbf{q}_i^k = \mathbf{A}_{Pi} \cdot \mathbf{d}_i^k + \mathbf{B}_{Pi} \cdot \mathbf{d}_{Si}^k$                 // M1
      7. $\mathbf{q}_{Si}^k = \mathbf{B}_{Pi}^{\mathbf{T}} \cdot \mathbf{d}_i^k + \mathbf{A}_{Si} \cdot \mathbf{d}_{Si}^k$                 // M2
      8. Update(..)                                             // M3
      9. $\tau^k = \text{InnerProduct}(\mathbf{d}_i^k, \mathbf{d}_{Si}^k \;;\; \mathbf{q}_i^k, \mathbf{q}_{Si}^k)$       // M4
      10. $\alpha^k = \gamma^k / \tau^k$                                     // M5
      11. $\mathbf{x}_i^{k+1} = \mathbf{x}_i^k + \alpha^k \mathbf{d}_i^k$;   $\mathbf{r}_i^{k+1} = \mathbf{r}_i^k - \alpha^k \mathbf{q}_i^k$        // M6
      12. $\mathbf{x}_{Si}^{k+1} = \mathbf{x}_{Si}^k + \alpha^k \mathbf{d}_{Si}^k$;   $\mathbf{r}_{Si}^{k+1} = \mathbf{r}_{Si}^k - \alpha^k \mathbf{q}_{Si}^k$     // M7
      13. ==                                                    // M8
      14. $\gamma^{k+1} = \text{InnerProduct}(\mathbf{r}_i^{k+1}, \mathbf{r}_{Si}^{k+1} \;;\; \mathbf{r}_i^{k+1}, \mathbf{r}_{Si}^{k+1})$   // M9
      15. If  $\sqrt{\gamma^{k+1}} \le \varepsilon$   STOP                          // M10
      16. $\beta^k = \gamma^{k+1} / \gamma^k$                                   // M11
      17. $\mathbf{d}_i^{k+1} = \mathbf{r}_i^{k+1} + \beta^k \mathbf{d}_i^k$;   $\mathbf{d}_{Si}^{k+1} = \mathbf{r}_{Si}^{k+1} + \beta^k \mathbf{d}_{Si}^k$     // M12
```

Figure 2. Algorithm for parallel solution of CGM

3.2 Hybrid Version

Theoretically, the hybrid program must offer a better performance than the pure message passing program for three reasons: 1) the message passing inside a node is replaced by a shared memory access (more fast); 2) the amount of communication decreases because the intra-node messages passing are not utilized; 3) there is a lesser amount of tasks involved in the communication, what leads a better scalability (Chow and Hysom, 2001).

To obtain a fine-grained hybrid version of the original MPI code, we applied an incremental approach. We first focused on parallelizing the loops on the parallel conjugate gradient method because this consumes about 90% of the application execution time. We parallelize the M1 and M2 modules in CGM algorithm (Fig. 2) because these modules are more expensive computationally and we verify the possibility to obtain some gains with the hybrid parallelization. Figure 3 shows the parallelization with OpenMP directives for the M1 and M2 modules in the CGM.

In the program code shown in Fig. 3, the **#pragma omp parallel** directive in line 3 specifies which code section must be run in parallel by the threads, where each thread is executed on one processor (line 4 to 32). The **#pragma omp for** directive, in line 5, specifies which iteration inside de loop started in line 6 must be split among the threads. Each thread runs a set of iterations in parallel with others. For this reason is necessary to verify if there are no data dependences inside the loop. The directive in line 19 does the same for the loop stared in line 20. In this paper we refer to tasks as units of processing. Here, not only MPI processes but also OpenMP threads are called tasks. Thus the comparisons have been made between executions with the same number of tasks in both models.

4. PERFORMANCE RESULTS

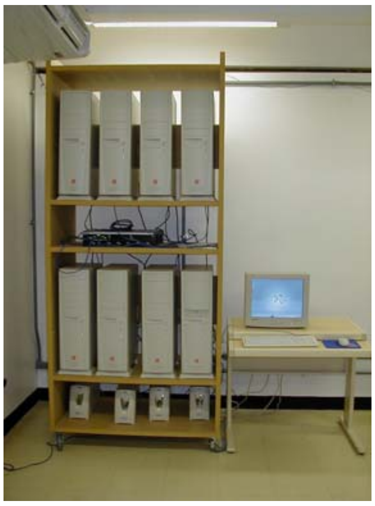
The target architecture is an eight node cluster (Fig. 4(a)), each node composed by a dual processor AMD Athlon MP 1900+ with 1.6 GHz mother board, 40GB of hard disk with access time of 10ms. Each node has 1.0GB of RAM memory (133MHz bus), 512KB of cache memory, and 1.0 GB of swap partition. The machines are interconnected using a fast Ethernet network of 1.0Gbps and the operating system installed is Linux Red Hat 9.0. The application has been compiled using the Intel 9.0 Compiler with OpenMP support. The MPI library used was the MPICH 7.2.

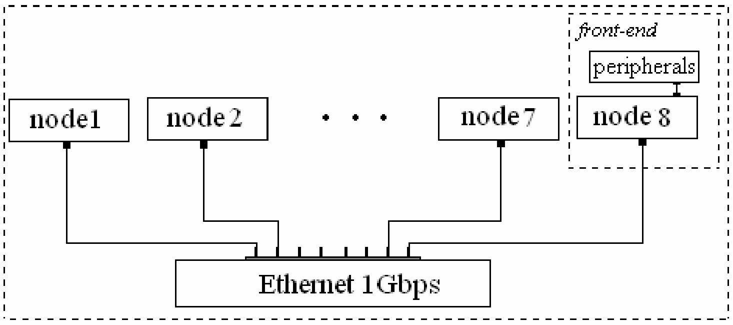
Fig. 4(b) shows a schematic representation of the cluster. Node eight is a complete machine with all his peripherals, and works as the front-end node controlling the cluster operation. In this node is installed the NFS (Network File System) for sharing data and programs, and the NIS (Network Information Service) assuring the access to all machines of the cluster. So, in the shared areas are stored the data input files, the font codes, communication libraries, and other necessary programs (e.g., METIS, CHACO, MPI, PVM, JiaJia).

The code has been instrumented using the RDTSC (Intel Corporation, 1996) instruction set and the total execution time has been decomposed in computation time and communication time. This decomposition is important to investigate our initial hypothesis that a hybrid solution would decrease the communication time and consequently decrease the computation time.

```
for (K = 0;  K< KMax; K++)
{
        #pragma omp parallel
        {
                # pragma omp for  nowait
                for (I = 0; I < p.glint; I++)
                {
                        A = B = 0;

                for (J = 0, Qp[ I ] = 0.0; J < p.glint, J++)
                                A+=Ap[ I ][ J ]*Pp[ J ];

                        for (J = 0; J < p.glfront; J ++)
                                B +=Bp[ I ][ J ]*Ps[ J ];

                Qp [ I ] = A+B;
                }

                # pragma omp for nowait
                for(I= 0; I < p.glfront; I++)
                {
                        C = D = 0;

                        for (J = 0, Qs[ I ] = 0.0; J < p.glint, J++)
                                C +=Bp[ I ] [ J ]*Pp [ J ];

                        for (J = 0; J < p.glfront; J ++)
                                D +=As[ I ][ J ]*Ps[ J ];

                        Qs [ I ] = C + D;
                }
        }
}
```

Figure 3. Hybrid code for parallel solution of CGM (M1 and M2 modules in Fig. 2)


(a)

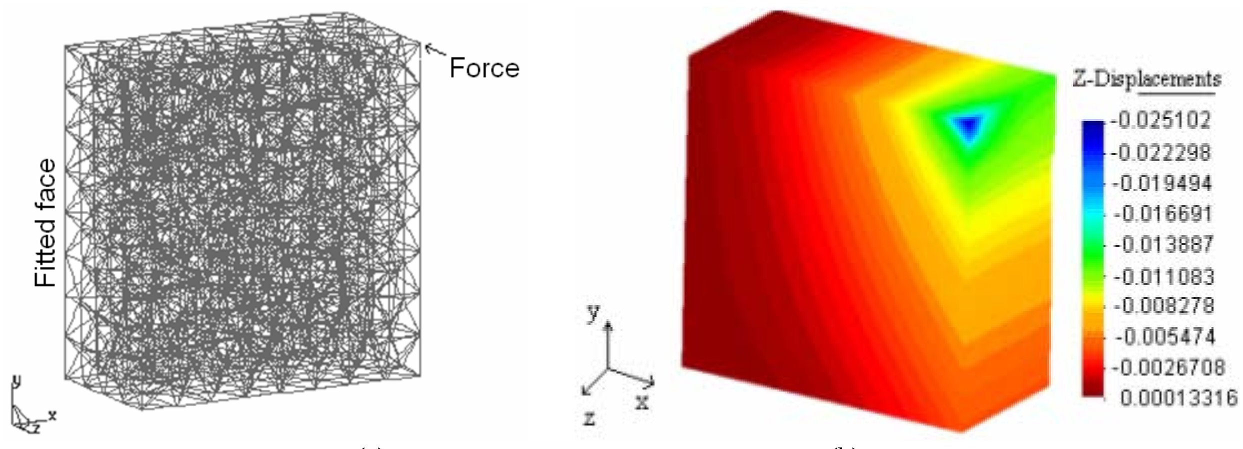


(b)

Figure 4. (a) Cluster (b) cluster assembly scheme

The geometry selected to our experiments is a block (1.0m × 1.0m × 0.5m) with on one of its faces rigidly fitted, and subjected to a bending force of 5.0 kN applied in the opposite direction of the z-axis, as shown in Fig 5(a). Young's modulus was considered to be E=21.0MPa and the Poisson coefficient $v = 0.2$.

The measures have been made for four 3D meshes detailed on Tab. 1. The mesh2 with 933 tetrahedrical elements is shown in Fig. 5(a). Figure 5(b) illustrates the displacement field in the *z* direction for the hybrid program. We verify that these results are very similar to pure program already validated.



(a) (b)

Figure 5. (a) Selected geometry: cantilever block with bending load in *z* direction
(b) displacement field in *z* direction (in meters)

Table 1. Meshes characteristics: number of nodes, elements and equations to be solved for the four meshes.

Mesh	Nodes	Elements	Equations
1	467	1882	1266
2	933	4014	2577
3	1984	9126	5567
4	3717	18031	10581

Figure 6 shows total time comparison for each of the meshes for hybrid and pure message passing versions of the application. We can verify that the hybrid model has a worst computation time in most cases, and the communication advantage does not compensate the computation disadvantage. However, we can also verify that the execution time for the hybrid version for many tasks running the smaller meshes was lesser. These occur because the execution time for these meshes is tiny and the communication time reduction is greater than the overhead introduced.

Figure 7 shows execution times for each one of the meshes, decomposed in computation time and communication time. The time results in Fig. 6 shows that in most cases the MPI total time is less than the hybrid one. On the other hand, in Fig. 7 we can see, as expected (as see in section 3.2), that in general the communication time on the pure version is higher than the communication time in the hybrid one.

Figure 8 shows observed speedup for hybrid and pure message passing versions of the application. The super linear speedup values indicating that the parallel solution is very efficient, better than predicted by Amdhal's law. For this tested case there are two important reasons that explain the speedup measured 1) lack of RAM memory. Since the total main memory in the cluster is larger than that in a single processor, and can hold more of the problem data at any instant, it leads to less, relatively slow disk memory traffic, 2) after the domain decomposition the original matrix with dimension $n\times n$ is divided between p parallel tasks. The partitioning looks for to divide the mesh in an homogeneous form, each task works on matrices with dimension $(n/p)\times(n/p)$, approximately. As the multiplication algorithm implemented in this code carries through $2\times(n/p)^2-(n/p)$ arithmetic operations per iteration of the CGM (M1 and M2 modules), the number of operations decreases with the square of the number of tasks.

We verify that the hybrid version has a good performance. The observed speedups in Fig 8 evidence this fact. However, in most situations, this performance is not better than the achieved by the MPI pure version. The bad results can be explained by the major weakness of the model. They are (Krawezik and Capello, 2003) (Capello and Etiemble, 2000): 1) the replication of the OpenMP parallel regions implies a high thread management cost. This factor is especially important in this application because we have used a fine grain approach. Besides that, because the application uses an iterative method to solve the equations, the threads must be forked several times. 2) Parallel regions lead to a bad utilization of the memory hierarchy.

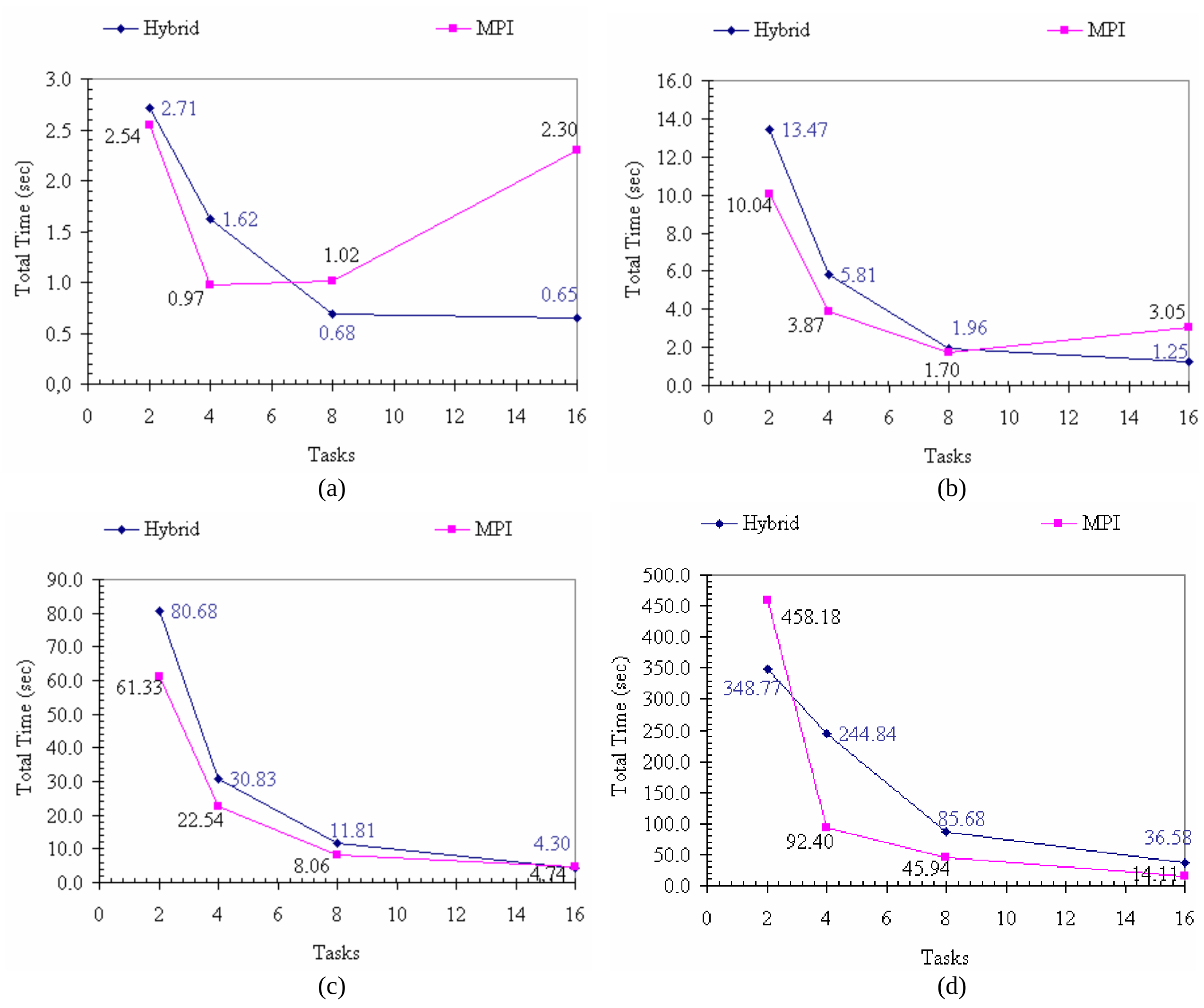


Figure 6. Total time for hybrid and MPI (pure) versions (a) mesh1, (b) mesh2, (c) mesh3, and (d) mesh4

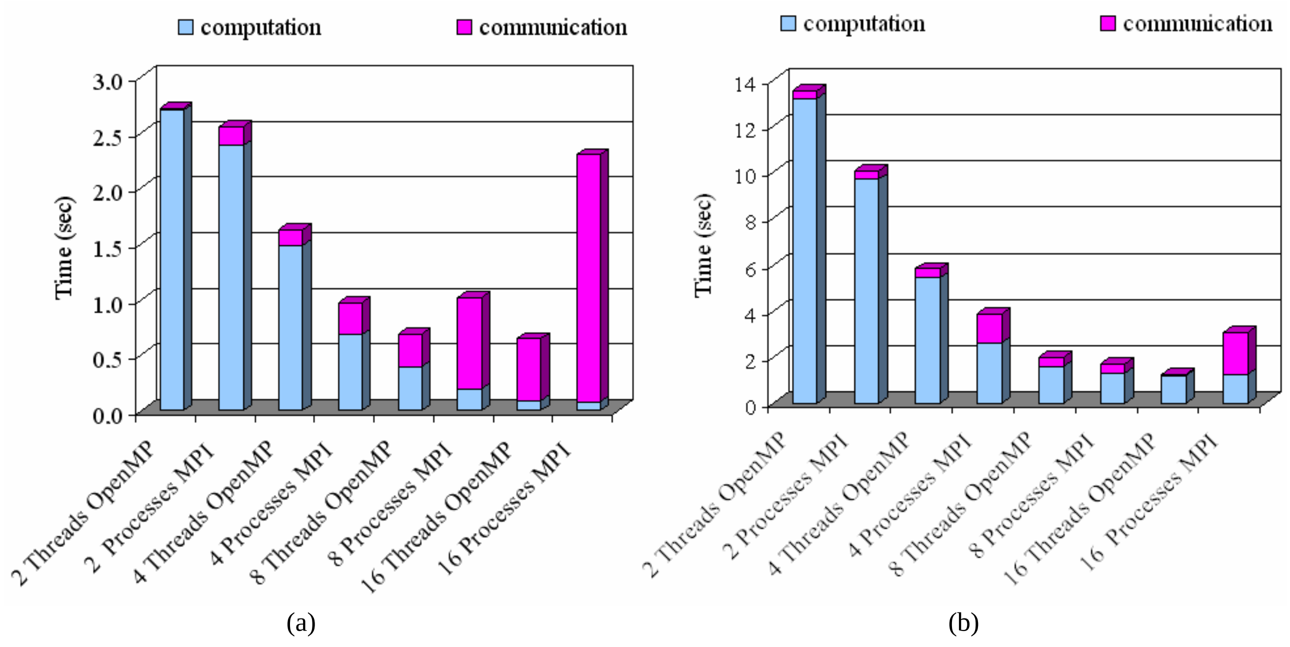
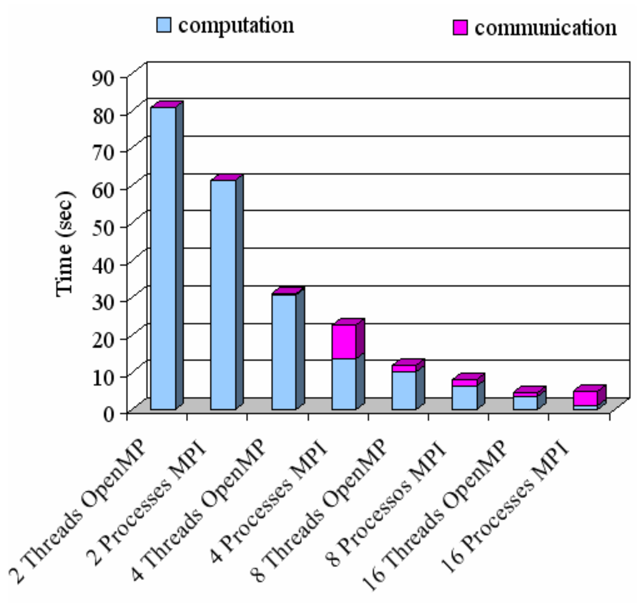
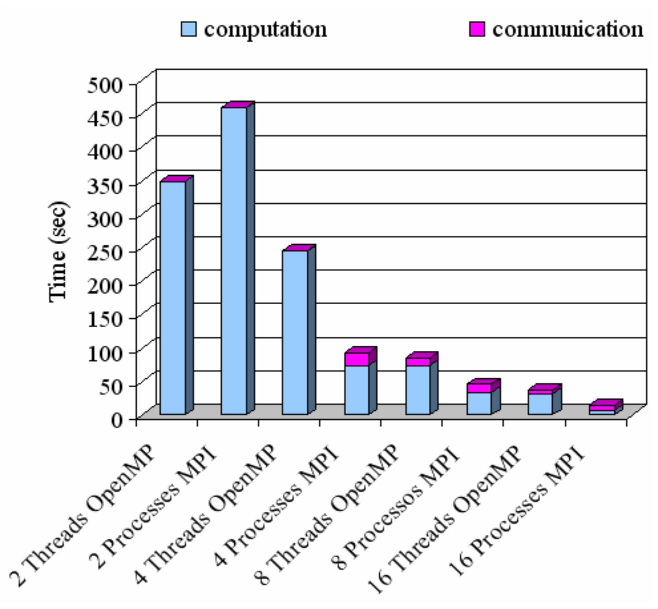


Figure 7. Computation and communication times: (a) mesh1, (b) mesh2

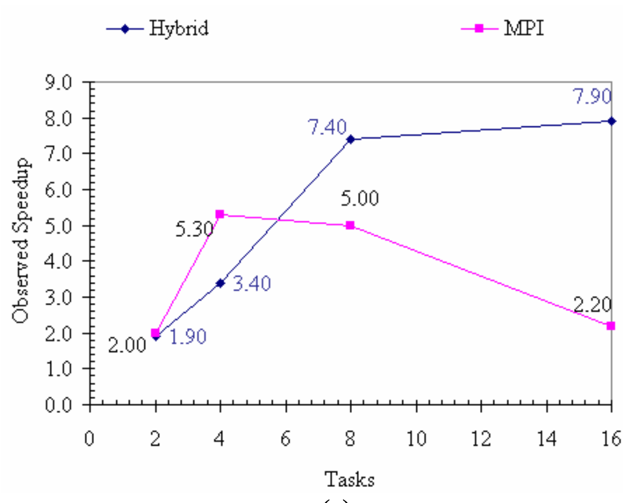


(c)

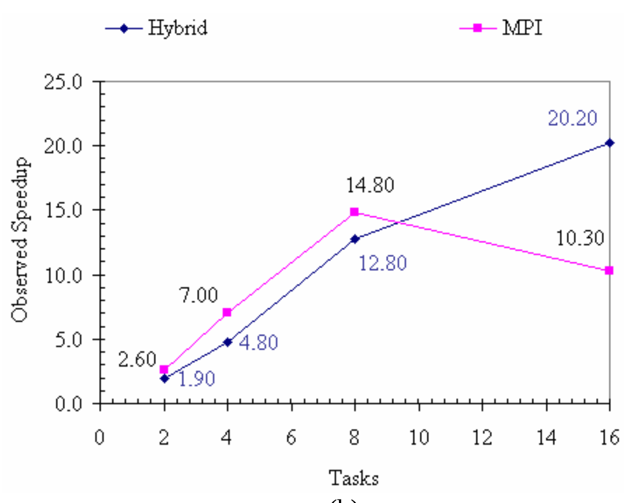


(d)

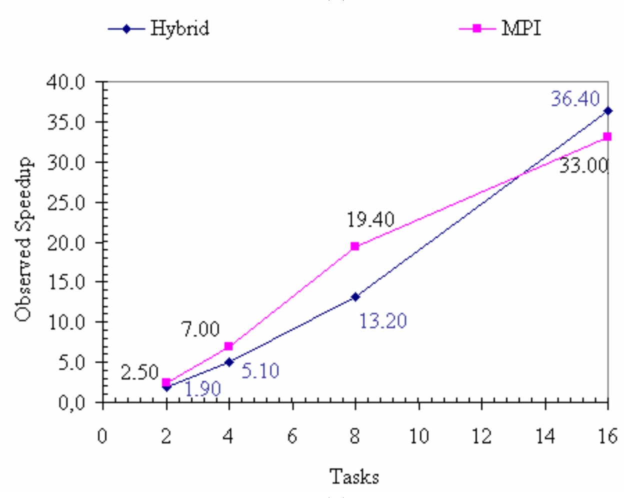
Figure 7. Computation and communication times: (c) mesh3, and (d) mesh4



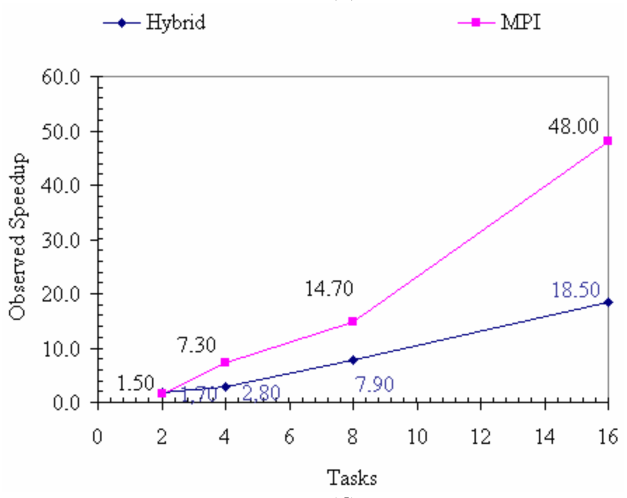
(a)



(b)



(c)



(d)

Figure 8. Observed speedup for (a) mesh1, (b) mesh2, (c) mesh3, and (d) mesh4

5. CONCLUSION

In this paper we developed and evaluated a hybrid model of parallel programming for a real engineering application based on the finite elements method. Hybrid models on SMP clusters have been used in several applications. Although some works like (Grabysz and Rabenseifner, 2005), (Loft et al, 2001) and (Viet et al, 2003) have reached better performance, most studies like (Capello and Etiemble, 2000) (Krawezik and Capello, 2003) (Chow and Hysom, 2001) come to a conclusion that the hybrid version loses to pure MPI versions in most cases. In (Capello and Etiemble, 2000) the author shows that the comparison results are clearly application dependent. However there are some optimizations like the one presented in (Viet et al, 2003) that can lead to very good results. Most of these optimizations, however, require a high programming effort and sometimes the original application must be completely rewritten to use them. In our results we have seen that in most cases the performance of the pure MPI model is better than the performance of the Hybrid model. However, we saw that with a hybrid model the communication overhead is reduced. This result shows that indeed, the use of a shared memory model inside the SMP node decreases the communication overhead. Thus, with some optimizations this technique can be very useful for some applications, especially those with intensive communication.

6. REFERENCES

Bathe, K.J., 1996, “Finite Element Procedures”, Ed. Prentice-Hall, Upper Saddle River, New Jersey, USA.

Cappello, F. and Etiemble, D., 2000, “MPI versus MPI-OpenMP on IBM SP for the NAS benchmarks”, Proceedings of the 2000 ACM/IEEE conference on Supercomputing, Dallas, Texas, USA.

Chow, E. and Hysom, D., 2001, “Assessing performance of hybrid MPI/OpenMP programs on SMP clusters”, Technical Report UCRL-JC-143957, Lawrence Livermore National Laboratory.

Cook, R. D., 1995, “Finite Element Modeling for Stress Analysis”, Ed. John Wiley, New York, USA.

Culler, D.E., Singh, J.P. and Gupta, A., 1999, “Parallel Computer Architecture: A Hardware/Software Approach”, Ed. Morgan Kaufmann Publisher, San Francisco, USA.

Dagum, L. and Menon, R., 1998, “OpenMP: An industry standard API for shared-memory programming”, J. IEEE Computational Science & Engineering, Vol. 5, No 1, pp. 46-55.

Gropp, W. and Lusk, E., 1995, “A taxonomy of programming models for symmetric multiprocessors and SMP clusters”, IEEE Programming Models for Massively Parallel Computers, pp. 2-7.

El-Rewini, H. and Lewis T. G., 1997, “Distributed and Parallel Computing”, Manning Publications Co, Greenwich, CT, USA.

George, K. and Vipin, K., 1988, “METIS – A Software Package for Partitioning Unstructured Graph, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices”, Manual, University of Minnesota, Department of Computer Science.

Grabysz, A., & Rabenseifner, R., 2005, “Nesting OpenMP in MPI to Implement a Hybrid Communication Method of Parallel Simulated Annealing on a Cluster of SMP Nodes”, Proceedings of the EuroPVM/MPI 2005, Lecture Notes in Computer Science, Sorrento, Italy, pp.18-21.

Hoeflinger, J.; 2006, “Extending OpenMP to Clusters”, Technical Report Intel Corporation.

Intel Corporation, 1996, “Using the RDTSC instruction for performance monitoring”, Technical Report Intel.

Krawezik, G. and Capello, F., 2003, “Performance Comparison of MPI and three OpenMP Programming Styles on Shared Memory Multiprocessors”, Proceedings of the fifteenth annual ACM symposium on Parallel algorithms and architectures, San Diego California, USA, pp. 118-127.

Loft, R., Thomas, S. and Dennis, J., 2001, “Terascale Spectral Element Dynamical Core for Atmospheric General Circulation Models”, Proceedings of 2001 ACM/IEEE conference on Supercomputing (CDROM), Denver, Colorado, USA, pp. 1-32.

Message Passing Interface Forum, 1993, “MPI: A Message Passing Interface”, International Journal of Supercomputer Applications, Vol. 15, No 19, pp. 878-883

Paramount Group, 2005, “Program Parallelization and Tuning Methodology”, at: <http://peak.ecn.purdue.edu/ParaMount/UMinor/methodology.html>.

Rabenseifner, R., 2003, “Hybrid Parallel Programming: Performance Problems and Chances”, Proceedings of the 45th CUG Conference, Columbus, Ohio, USA, pp.12-16.

Schäfer, M., 2006, “Computational Engineering. Introduction to Numerical Methods”, Ed. Springer, Berlin, 326 p.

Smith, L. and Bull M., 2001, "Development of Mixed Mode MPI / OpenMP Applications", Scientific Programming, Vol. 9, No 2-3, pp. 83-98.

Viet, T., Yoshinaga, T., Abderazek, B.A. and Sowa, M., 2003, “A Hybrid MPI-OpenMP Solution for a Linear System on a Cluster of SMPs”, Proceedings of Symposium on Advanced Computing Systems and Infrastructures, Japan, pp.299-306.

Villa Verde, F. R, Pfitscher, G. H. and Viana, D. M., 2004a, “Paralelização em ambientes PVM e MPI de uma aplicação do método dos elementos finitos”, Proceedings of III Congresso Nacional de Engenharia Mecânica, Belém, Pará, Brazil.

Villa Verde, F. R., Pfitscher, G. H., Viana, D. M. and Kummel, B. C., 2004b, “Análise Comparativa da Execução Paralela em PVM e MPI de Uma Aplicação do Método dos Elementos Finitos”, Proceedings of Iberian Latin American Congress on Computational Methods in Engineering, Recife, Pernambuco, Brazil.

Villa Verde, F. R.; Pfitscher, G. H. Viana, D. M., 2004c, “Expressive Gains in Performance on Parallel Solution of Finite Element Method Applications Using Low Cost PC Cluster”, Procedure of I2TS, São Carlos, São Paulo, Brazil.

Villa Verde, F. R., Pfitscher, G. H. and Viana, D. M., 2005, “Parallel Solution on a PC Cluster and Behavior Analysis of Bi and Three Dimensional Structural Problems Utilizing the Finite Element Method”, Proceedings of the International Congress of Mechanical Engineering, Ouro Preto, Minas Gerais, Brazil.

Villa Verde, F. R. and Pfitscher, G. H., 2005, “Effects of data granularity and memory size on simulations in pc cluster of a linear elasticity problem modeled by finite elements”, Proceedings of the XXVI Iberian Latin American Congress on Computational Methods in Engineering, Guarapari, Espirito Santo, Brazil.

Villa Verde, F. R., Pfitscher, G. H. and Viana, D. M., 2006, “Simulação de problemas estruturais aplicados à elasticidade linear em agregados de PCs”, Proceedings of IV Congresso Nacional de Engenharia Mecânica, Recife, Pernambuco, Brazil.

Villa Verde, F. R., Pfitscher, G. H., Viana, D. M. and Ferreira, J.L., 2007, “Caracterização de Desempenho de Códigos Paralelos que Utilizam Estratégia de Decomposição de Domínio em Clusters de PCs ”, Proceedings of the XXVIII Iberian Latin American Congress on Computational Methods in Engineering, Porto, Portugal.

7. RESPONSIBILITY NOTICE

The authors are the only responsible for the printed material included in this paper.